MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 89-538
May 1989

Redundancies and Don't Cares in Sequential Logic Synthesis

Srinivas Devadas, Hi-Keung Tony Ma, and A. Richard Newton

Abstract

The relationships between redundant logic and don't care conditions in combinational circuits are well known. Redundancies in a combinational circuit can be explicitly identified using test generation algorithms or implicitly eliminated by specifying don't cares for each gate in the combinational network and minimizing the gates, subject to the don't care conditions.

In this paper, we explore the relationships between redundant logic and don't care conditions in sequential circuits. Stuck-at faults in a sequential circuit may be testable in the combinational sense, but may be redundant because they do not alter the *terminal behavior* of anon-scan sequential machine. The *sequential redundancies* result in a faulty State Transition Graph (STG) that is equivalent to the STG of the true machine.

We present a classification of redundant faults in sequential circuits composed of single or interacting finite state machines. For each of the different classes of redundancies, we define don't care sets which if optimally exploited will result in the implicit elimination of any such redundancies in a given circuit. We present systematic methods for the exploitation of *sequential don't cares* that correspond to *sequences* of vectors that never appear in cascaded or interacting sequential circuits. Using these don't care sets in an optimal sequential synthesis procedure of state minimization, state assignment and combinational logic optimization result in *fully testable* lumped or interacting finite state machines. We present experimental results which indicate that irredundant sequential circuits can be synthesized with *no area overhead* and within reasonable CPU times by exploiting these don't cares.

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

To be presented at the *International Test Conference*, Washington, D.C., Aug. 1989. This work was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-87-K-0825.

Author Information

Devadas: Department of Electrical Engineering and Computer Science, Room 36-848, MIT, Cambridge, MA 02139. (617) 253-0454.
Ma and Newton: Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

Redundancies and Don't Cares in Sequential Logic Synthesis

Srinivas Devadas
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, Cambridge

Hi-Keung Tony Ma and A. Richard Newton
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

The relationships between redundant logic and don't care conditions in combinational circuits are well known. Redundancies in a combinational circuit can be explicitly identified using test generation algorithms or implicitly eliminated by specifying don't cares for each gate in the combinational network and minimizing the gates, subject to the don't care conditions.

In this paper, we explore the relationships between redundant logic and don't care conditions in sequential circuits, extending the results of [5]. Stuck-at faults in a sequential circuit may be testable in the combinational sense, but may be redundant because they do not alter the *terminal behavior* of a non-scan sequential machine. These *sequential redundancies* result in a faulty State Transition Graph (STG) that is equivalent to the STG of the true machine.

We present a classification of redundant faults in sequential circuits composed of single or interacting finite state machines. For each of the different classes of redundancies, we define don't care sets which if optimally exploited will result in the implicit elimination of any such redundancies in a given circuit. We present systematic methods for the exploitation of *sequential don't cares* that correspond to *sequences* of vectors that never appear in cascaded or interacting sequential circuits. Using these don't care sets in an optimal sequential synthesis procedure of state minimization, state assignment and combinational logic optimization results in *fully testable* lumped or interacting finite state machines. We present experimental results which indicate that irredundant sequential circuits can be synthesized with *no area overhead* and within reasonable CPU times by exploiting these don't cares.

1 Introduction

The connection between logic minimization and test generation for combinational circuits is well known and has been systematically investigated [2] [1]. The connection rests on the simple observation that the absence of a test is associated with *redundancy* in the logic network.

The sources of redundancy in combinational circuits are well understood and algorithms are available for making representations of incompletely specified combinational logic functions irredundant. The algorithms in [1] determine a don't care set for each gate in a multi-level network and minimize the logic function corresponding to the gate, subject to these don't care conditions to produce an irredundant, fully testable circuit. Combinational test generation algorithms (e.g. [8], [11]) can be used to explicitly identify redundancies in a logic circuit, which can then be eliminated.

Redundancies in sequential machines may alter the functionality of the combinational logic in the (non-scan) sequential machine, while maintaining the *terminal behavior* of the machine. Thus, a stuck-at fault may be testable from a combinational point of view, but may be *sequentially redundant*. This redundant fault results in a faulty State Transition Graph (STG) that is equivalent to the true STG.

Sequential test generation and sequential logic synthesis algorithms are much less developed than their combinational counterparts. The observation that the absence of a test is associated with redundancy holds for sequential circuits as well. While test generation algorithms can conceivably be used to produce irredundant sequential circuits, by explicitly identifying and eliminating redundancy, such an approach would require astronomical CPU time for anything but the smallest circuits, given the efficiency of state-of-the-art sequential test generation algorithms.

The relationships between don't care conditions and redundancies in sequential circuits are not well understood at present. Intuitively, one would expect that sequential redundancies are intimately related to representations of don't cares in the various steps of sequential logic synthesis, namely, state minimization, state assignment and combinational logic optimization.

In [7] and [4], constrained synthesis procedures that result in fully and easily testable finite state machines were presented. These procedures effectively involve the addition of extra logic to produce an easily testable machine. In [5], a classification of redundancies in sequential circuits composed of a *single* finite state machine was presented. An optimal synthesis procedure that results in a *fully testable* sequential machine was given.

In the work presented in [5], each of the different classes of redundancies in single FSMs was shown to be tied to a don't care set, which if optimally exploited results in the implicit elimination of that form of redundancy. The work in [5] was restricted to single finite state machines (FSMs). *Multi-level sequential circuits* composed of interacting finite state machines, are common in industrial chip designs and are more complicated. The types of possible redundancies and their corresponding don't care sets in cascaded/interconnected FSMs are likewise more complicated.

We show in the cascaded machine case that *sequential don't cares*, corresponding to sequences of vectors that never occur, at the inputs (outputs) of the driven (driving) FSM have to be used to optimize the driven (driving) machine, in order to eliminate certain kinds of redundancies. We present systematic methods of exploiting sequential don't cares in cascaded circuits. Interacting finite state machines can be viewed as separate occurrences of cascades and don't care sets can be iteratively used on the different cascades to eliminate redundancy. Finally, we present experimental results which indicate that *fully testable* interacting FSMs can be produced from State Transition Graph descriptions via optimal sequential logic synthesis with reasonable CPU time expenditure.

In the next section, basic definitions and notations used are given. In Section 3, we classify redundancies in single finite state machines and the don't care sets required to eliminate these redundancies. We present new algorithms for the systematic exploitation of sequential don't cares in cascaded circuits in Section 4. The implicit elimination of redundancies in cascaded finite state machines is the subject of Section 5. Iterative optimization procedures for interacting FSMs are discussed in Section 6. Preliminary experimental results are presented in Section 7.

2 Preliminaries

A **variable** is a symbol representing a single coordinate of the Boolean space (e.g. a). A **literal** is a variable or its negation (e.g. a or $\overline{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (e.g., $\{a, b, \overline{c}\}$ is a cube, and $\{a, \overline{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively. An **expression** is a set f of cubes. For example, $\{\{a\}, \{b, \overline{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \overline{c}\}$. An expression represents the disjunction of its cubes.

A cube may also be written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries.

A finite state machine is represented by its **State Transition Graph** (STG), $G(V, E, W(E))$ where V is the set of vertices corresponding to the set of states S, where $||S|| = N_s$ is the cardinality of the set of states of the FSM, an edge joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j, and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused that transition and the values of the primary outputs corresponding to that transition. In general, the $W(E)$ labels are Boolean expressions. The number of inputs and outputs are denoted N_i and N_o respectively. The input combination and present state corresponding to an edge or set of edges is (i, s), where i and s are cubes. The fanin of a state q is a set of edges and is denoted $fanin(q)$. The fanout of a state q is denoted $fanout(q)$. The output and the fanout state of an edge $(i, s) \in E$ are $o((i, s))$ and $n((i, s)) \in V$ respectively.

Given N_i inputs to a machine, 2^{N_i} edges with minterm input labels fan out from each state. A STG where the next state and output labels for every possible transition from every state are defined corresponds to a **completely specified machine**. An **incompletely specified machine** is one where at least one transition edge from some state is not specified.

A starting or initial state is assumed to exist for a machine, also called the **reset state**. Given a logic-level finite state machine with N_k latches, 2^{N_k} possible states exist in the machine. A state which can be reached from the reset state via some input vector sequence is called a **valid state** in the STG. The input vector sequence is called the **justification sequence** for that state. A state for which no justification sequence exists is called an **invalid state**. Given a fault F, the State Transition Graph of the machine with the fault is denoted G^F. Two states in a State Transition Graph G are **equivalent** if all possible input sequences when the machine is initially in either of the two states produce the same output response.

A State Transition Graph G_1 is said to be **isomorphic** to another State Transition Graph G_2 if and only if they are identical except for a renaming of states.

The fault model assumed is **single stuck-at**. A finite state machine is assumed to be implemented by combinational logic and feedback registers. Tests are generated for stuck-at faults in the combinational logic part.

A primitive gate in a network is **prime** if none of its inputs can be removed without causing the resulting circuit to be functionally different. A gate is **irredundant** if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be **prime** if all the gates are prime and **irredundant** if all the gates are irredundant. It can be shown that a gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults.

We differentiate between two kinds of redundancies in a sequential circuit. If the effect of the fault cannot be observed at the primary outputs or the next state lines, beginning from any state, with any input vector, the fault is deemed **combinationally redundant**. A **sequentially redundant** fault is a fault that cannot be detected by any input sequence and is not combinationally redundant.

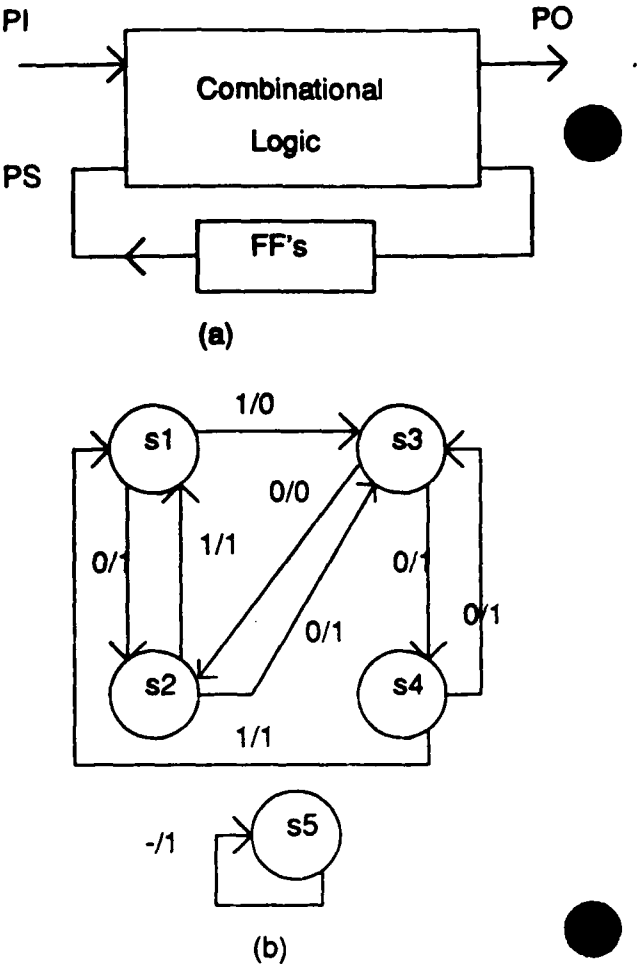


Figure 1: A Sequential Circuit

To detect a fault in a sequential machine, the machine has to be placed in a state which can then excite and propagate the effect of the fault to the primary outputs. The first step of reaching the state in question is called **state justification**. The second step is called **fault excitation-and-propagation**.

An edge in a State Transition Graph of a machine is said to be **corrupted** by a fault if either the fanout state or output label of this edge is changed because of the existence of the fault. A path in a State Transition Graph is said to be corrupted if at least one edge in the path has been corrupted.

A sequence of vectors VS_1 is said to **contain** another sequence VS_2 (written as $VS_1 \supseteq VS_2$), if VS_2 appears in VS_1.

A **cascade** of two machines A and B is denoted $A \longrightarrow B$. A is the **driving** machine and B the **driven** machine.

3 Redundancies in Single Finite State Machines

A sequential circuit S, comprised of a single FSM is shown in Figure 1(a). The State Transition Graph corresponding to the circuit is shown in Figure 1(b).

Redundant faults in S may be combinationally redundant (CRFs) or sequentially redundant (SRFs). Sequentially redundant faults can be classified into three categories.

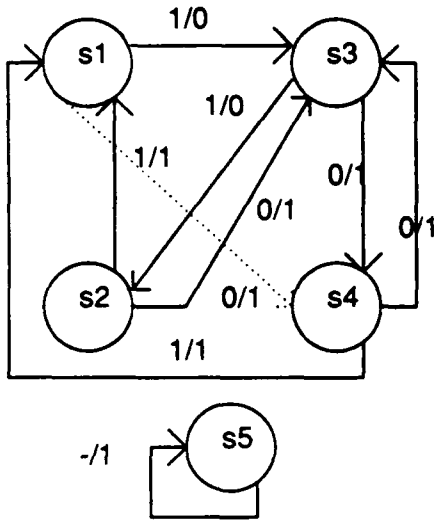


Figure 2: A Type 1 SRF

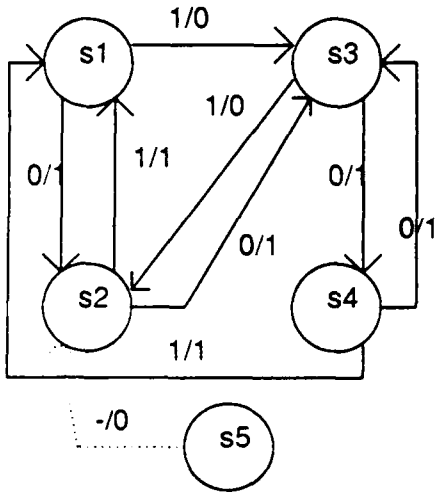


Figure 3: A Type 2 SRF

1. The fault causes the interchange/creation of equivalent states in the STG.

2. The fault does not corrupt any fanout edge of a valid state in the STG.

3. The fault results in a faulty machine that is isomorphic (with a different encoding) to the original machine.

In [5], it was shown that no other kind of sequential redundancy could exist.

In Figure 1(b), states *s2* and *s4* are equivalent states. A *type 1 SRF* in S may produce the faulty STG of Figure 2, where the only corrupted edge (shown in dotted lines) goes to *s4* instead of *s2* and does not change the terminal behavior of S. A faulty STG corresponding to a *type 2 SRF* is shown in Figure 3. Only fanout edges from an invalid state have been corrupted. In Figure 4, an isomorphic faulty machine (equivalent to the true machine) is shown where *s2* and *s3* have been interchanged.

3.1 Eliminating Type 3 SRFs

In [5], it was shown that stuck-at faults in a sequential machine implemented by a two-level combinational network could not cause isomorphism. For a sequential machine, implemented by a multi-level network, stuck-at faults could conceivably produce an isomorphic faulty STG.

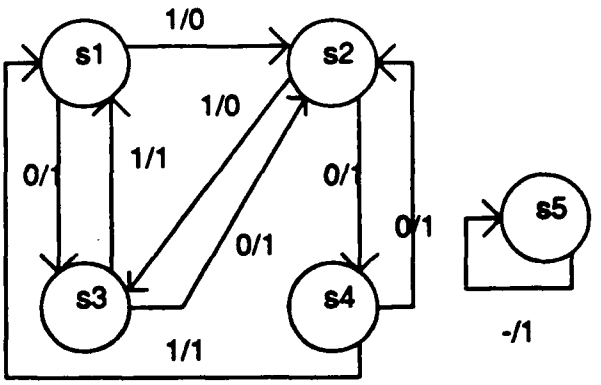


Figure 4: A Type 3 SRF

There are many ways of ensuring that isomorphism does not occur in multi-level networks. Isomorphism due to a fault is essentially due to a *sub-optimal state assignment*. The new encoding corresponding to the isomorph represents a better machine (one with the redundant line removed). A locally optimal state assignment across any given set of states can ensure that isomorphism does not occur in multi-level circuits, across this set of states. It is worthwhile to note that optimal state assignment corresponds to the optimal usage of don't cares – one does not care what the codes of the different states are so long as they are distinct.

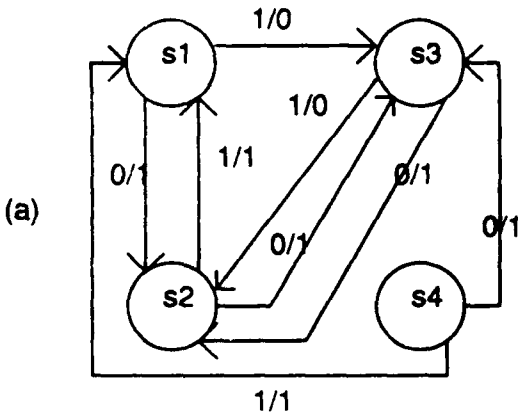
3.2 Eliminating Type 2 SRFs

The codes corresponding to invalid states can be used as don't cares during logic optimization. A type 2 SRF is due to the sub-optimal usage (or no usage) of these don't cares. These redundancies will not exist if the combinational logic is made irredundant under this don't care set.

3.3 Eliminating Type 1 SRFs

Type 1 SRFs are related to redundant states in a sequential machine. Given a reduced machine, a fault that corrupts a single edge going to a faulty but valid state cannot be redundant, since all states are distinguishable. Thus, an initial state minimization will preclude the occurrence of the SRF of the form in Figure 2. However, we may have a case where the fault results in a faulty invalid next state that is equivalent to the true next state. This is illustrated in Figure 5. We have the true STG in Figure 5(a), that is state minimal. The invalid state *s4*'s code has been used as a don't care and *s4* is equivalent to state *s2* after logic minimization under this don't care condition. A fault could result in the scenario shown in Figure 5(b), where a single corrupted edge whose true next state is *s2* produces a faulty next state, *s4*. The fault is redundant. This redundancy exists because we have not exploited the don't care corresponding to the edge (0, *s3*) – we can specify $n(0, s3) = (s4, s2)$ and not just *s2*. The following procedure of repeated logic minimization (modified from [5]) guarantees upon convergence that *type 2 SRFs* don't exist and that single edge corrupting and certain kinds of multiple edge corrupting *type 1 SRFs* don't exist. The use of extended don't cares at Step **A** guarantees the elimination all possible of *type 1 SRFs*, but these don't cares are *not* required in practice to produce irredundant machines.

eliminate-type1/2-SRFs(S):
{
  iter = 1 ;
  if (*iter* = 1) G = **extract-stg**(S) ;



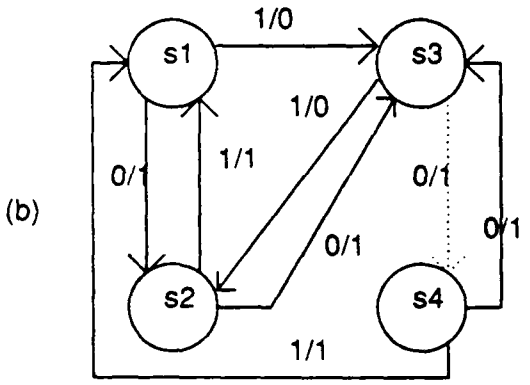


Figure 5: A Complex Type 1 SRF

```
  else G = extract-stg( S'' ) ;
  do {
    foreach ( valid state q ∈ G ) {
      Find all valid states (v_1, .. v_m) ≡ q ;
      Find all invalid states (iv_1, .. iv_n) ≡ q ;
      A: FA^DC | fanin(q) = (q, v_1, .. v_m, iv_1, .. iv_n) ;
    }
    S' = optimize( S, FA^DC ) ;
    IV = extract-invalid-states( S' ) ;
    S'' = optimize( S', IV^DC ) ;
    iter = iter + 1 ;
  } while( S ≠ S'' ) ;
}
```

It can be proved that state minimization, a locally optimal state assignment and the procedure **eliminate-type1/2-SRFs**() produces an irredundant sequential machine [5].

4 Exploiting Sequential Don't Cares

In Figure 6, we have a machine A driving another machine B via a set of latches $L1$ (We neglect C for the moment). For the purposes of the discussion here, we assume that all the latches in $L1$ are not observable. In practice, a subset of the latches may be observable.

4.1 Don't Care Inputs for the Driven Machine

There are several don't care conditions associated with the intermediate lines corresponding to $L1$, which are inputs to

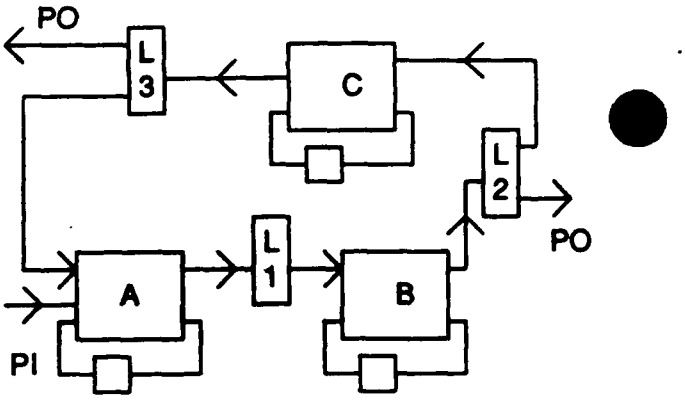


Figure 6: Interacting Finite State Machines

B. Let the number of intermediate/pipeline latches in $L1$ be N.

1. A may or may not assert all 2^N possible output combinations. If a certain binary combination, c_1 never appears at $L1$, then B can be made incompletely specified – the transition edges corresponding to an input of c_1 need not be specified, whatever state B is in (We don't care what happens when B receives the input c_1).

2. A more general case of (1) is when a certain combination c_2 never appears at $L1$, when B is in some set of states $Q_B \in S_B$. It does appear when B is in states other than Q_B. In this case, the states in Q_B will have c_2 unspecified (If an edge on c_2 exists in Q_B, it can be removed).

3. A more complicated sequential don't care is associated with vector *sequences* that never appear at $L1$, though all 2^N separate vectors appear. A does not produce all possible output sequences. This type of don't care does *not* have a straightforward interpretation. Edges in the State Transition Graph of B cannot be removed or left unspecified directly.

Both (1) and (2) can be easily exploited via the use of standard state minimization algorithms that handle incompletely specified machines [10]. However, exploiting the don't care input *sequences* is more complicated and systematic methods have not been proposed to date.

In Figure 7, a State Transition Graph corresponding to a possible B machine is shown. The machine is state minimal. We assume that each transition edge in B is irredundant, i.e. B makes every transition with appropriate input sequences. A don't care input sequence is shown below the Graph. Such a don't care sequence implies that certain ***sequences of transitions*** will not be made by B.

A don't care input sequence is assumed to have a length greater than 1. Given a don't care sequence DC, all sequences SE such that $SE \supseteq DC$ are also don't care sequences. We define an **atomic don't care sequence** as one that does not contain any other don't care sequence. Thus, any subsequence of an atomic don't care sequence is a care sequence. In the sequel, we consider only atomic don't care sequences.

Our problem lies in exploiting this form of don't care, so as to optimize B. In the general case, we will have a set of don't care sequences. We can state the following lemma.

Lemma 4.1 : *Given a machine B and a set of don't care sequences DC_j , $1 \le j \le N_C$, if two states in B, s1 and s2 have distinguishing sequences I_i , $1 \le i \le N_D$ such that for each k, $I_k \supseteq DC_l$ for some l, then s1 and s2 are equivalent in B under the DC_j.*

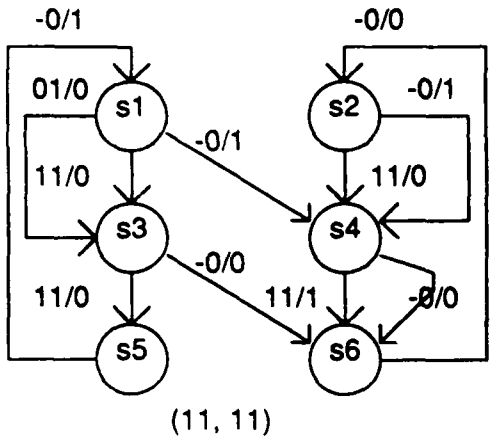


Figure 7: Input Don't Care Sequences

Proof: Since the DC_j can never occur, it means the I_i can never occur. Therefore, $s1$ and $s2$ in B are equivalent under DC_j. **Q.E.D.**

An approach to exploit don't cares based on Lemma 4.1 would entail producing all distinguishing sequences for every pair of states in B and checking for the containment condition. Pairs satisfying the condition can be merged. This is potentially very time consuming: a pair of states may have many distinguishing sequences and we have to find them for every possible pair. A more efficient approach is now outlined.

In this approach, given a set of don't care sequences, B is transformed into a new machine B' which has a greater number of states, but is more incompletely specified than B. B' is state minimized to obtain B'' ($||S_{B''}|| \leq ||S_B||$). The pseudo-code below illustrates the procedure.

```
exploit-input-dc( B, DC ):
{
  B' = B ;
  foreach ( don't care sequence DC_i ) {
    foreach (depth-first path P = e_1, .. e_K  ∈ B' ) {
      if ( P ⊇ DC_i ) {
        for( i = 2; i ≤ K; i = i + 1 ) {
          s_i = e_i- > fanout ;
          make states s'_i and s''_i ;
          fanin(s'_i) = e_{i-1} ;
          fanin(s''_i) = fanin(s_i) - e_{i-1} ;
          if ( fanin(s''_i) = φ ) delete s''_i ;
          if ( i < K )
            fanout(s'_i) = fanout(s''_i) = fanout(s_i) ;
          else {
            fanout(s'_i) = fanout(s_i) - e_{i-1} ;
            fanout(s''_i) = fanout(s_i) ;
          }
          delete s_i ;
        }
      }
    }
  }
  B'' = state-minimize ( B' ) ;
}
```

The procedure is effectively producing a machine where the don't care sequences are *not* specified, but otherwise has the same functionality as the original machine. This means that if any two states in B satisfy the conditions of Lemma 4.1, these two states will not possess a distinguishing sequence in B' and will thus be *compatible* during state minimization. A smaller machine B'' will be obtained after state minimization.

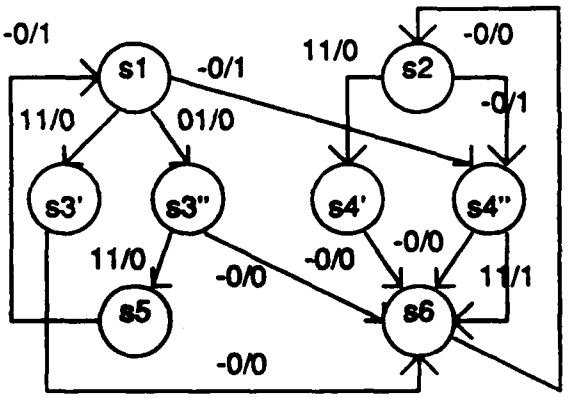


Figure 8: Expanding the Original Machine

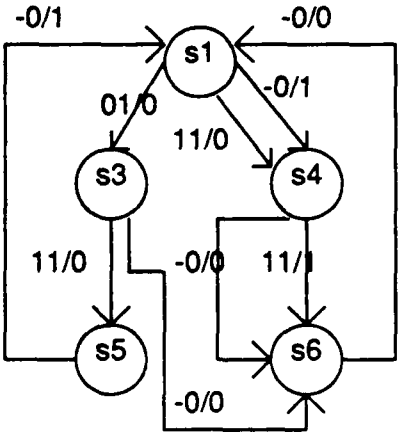


Figure 9: State minimization after Expansion

When $i = p < K$ in the *for* loop above, the fanout of s_p is duplicated for the states s'_p and s''_p – the edge e_p is also duplicated. Hence, at the next iteration, one of the e_p fans into s'_{p+1} and the other e_p (as well as the remaining fanout edges from s'_p and s''_p) into s''_{p+1}.

An illustrative example is given in Figures 8 and 9. The machine and the don't care sequence of Figure 7 produce an expanded machine, shown in Figure 8. State minimizing this machine produces the result of Figure 9, which has one less state than the original machine of Figure 7.

Given a cascade, we need to generate the set of sequences that the driving machine in a cascade $A \longrightarrow B$ never asserts, so as to optimize the driven machine B as in Figures 8 and 9. This is done by generating don't care sequences of increasing length, beginning from a length of 2. Starting from each possible state in A, all possible 2-vector sequences are found. Single vectors that don't occur are added to this set and the set is "complemented" to find the atomic 2-vector sequences that don't occur. Next, all sequences of length 3 that A asserts are found. The single-vector and 2-vector don't care sequences are added to this set and the union is complemented to find the atomic don't care sequences of length 3.

4.2 Don't Care Outputs for the Driving Machine

The sequential don't cares discussed thus far are a product of the constrained controllability of the driven machine B in a cascade $A \longrightarrow B$. There is another type of don't care due to the constrained observability of the driving machine A. We

i1	sa1	sa2	INT1	INT1	qb1	qb2	out1
i2	sa1	sa3	INT2	INT2	qb1	qb3	out2
i1	sa2	sa1	INT2	INT1	qb2	qb2	out3
i2	sa2	sa3	INT1	INT2	qb2	qb2	out3
i1	sa3	sa1	INT1	INT1	qb3	qb2	out4
i2	sa3	sa2	INT1	INT2	qb3	qb3	out1

A ⟶ B

Figure 10: Output Expansion

focus on the individually state minimized tables of Figure 10. The intermediate inputs/outputs have been given symbolic codes. Given that A feeds into B, it is quite possible that for some transition edge $e_1 \in A$, it does not matter if the output asserted by this particular transition edge is, say, $INTi$ or $INTj$. In fact, in Figure 10, the 3rd transition edge can be either $INT1$ or $INT2$, *without changing the terminal behavior of* $A \longrightarrow B$. This is a don't care condition on A's outputs. It is quite possible that making use of these don't cares can reduce the number of states in A. In fact, if one replaced the output of the 3rd edge in A (Figure 10) by $INT1$ instead of $INT2$, we would obtain one less state after state minimization. (*sa2* becomes equivalent to *sa3*).

Given a cascade $A \longrightarrow B$, we give below a systematic procedure to detect this type of don't care, i.e. expand the output of each transition edge of A to the set of all possible values that it can take while maintaining the terminal behavior of $A \longrightarrow B$. Standard state minimization procedures can exploit don't care outputs, represented as cubes. However, state minimization procedures have to be modified in order to exploit transition edge outputs represented as arbitrary Boolean expressions (multiple cubes).

```
output-expansion-1( A, B ):
{
  foreach ( edge e1 ∈ A ) {
    OUT(e1) = universe ;
    foreach ( state q1 ∈ QB ) {
      if ( B can be in q1 when A makes transition e1 ) {
        find largest set of output combinations
        c1 ∃ c1 ⊇ e1-> output &&
          fanin(c1, q1), output(c1, q1) are unique ;
        OUT(e1) = OUT(e1) ⋂ c1 ;
      }
    }
    e1-> output = OUT(e1) ;
  }
  A = state-minimize( A ) ;
}
```

A transition edge e_1 in A is picked. The set of states that B can be in when A makes this transition is found. Given this set of states, the largest cube (or set of output combinations) that covers the output of the edge and produces a unique next state and a unique output when B is in *any* one of the possible states is found (corresponds to $OUT(e_1)$). The output of e_1 is expanded to the cube. The process is repeated for all edges in A.

The state minimization procedure proposed in [10] can be used for incompletely specified finite state machines. However, after output expansion, we may have a multiple-output FSM in which a transition edge has an output that can belong to a subset of symbolic or binary values, rather than the universe of possible values (as in the incompletely specified case).

In the state minimization procedure of [10], two states are deemed to be compatible if the output combinations that can be asserted by each pair of corresponding fanout edges of the two states intersect. One can envision a situation where the possible output combinations of the fanout edges of $q_1, q_2 \in S_M$ intersect leading to a compatibility relation $q_1 \leftrightarrow q_2$, with similar compatibility relations $q_2 \leftrightarrow q_3$ and $q_1 \leftrightarrow q_3$. However, the three-way intersection between the possible output combinations of the fanout edges of q_1, q_2 and q_3 may be a null intersection, implying that q_1, q_2 and q_3 cannot be merged into a single state, even though all the required pairwise compatibility relations exist. In the binary-valued output case, if the possible output combinations can be represented as a single cube, then such a situation will not occur, since the three-way intersection of a set of three cubes has to be non-empty if the pairwise intersections are non-empty. But, in the case of multiple cubes or Boolean expressions specifying the output combinations for fanout edges, an additional check has to be performed during state minimization during the selection of the compatibility pairs to see if three or more sets of states can, in fact, be merged, preserving functionality.

5 Fully Testable Cascaded Finite State Machines

In this section, we present a classification of redundant faults in a cascade. We define don't care sets tied to each of these forms of redundancy and give a synthesis procedure that produces an irredundant cascade.

5.1 Redundancies in a Cascade

Redundancies in a cascade $A \longrightarrow B$ can be classified into four categories. The intermediate lines are denoted INT.

1. $F \in A$ that cannot propagate to the intermediate lines INT.
2. $F \in A$ that propagates to INT but not PO, the primary outputs of B.
3. $F \in B$ that does not propagate to PO, but would have if INT were completely controllable.
4. $F \in B$ that does not propagate to PO and would not have even if INT were completely controllable.

Obviously, there can be no other class of redundancy.

It is easy to see that redundancies of type (1) and (4) are associated with the single machines A and B. If A and B are irredundant, these redundancies will not appear in a cascade $A \longrightarrow B$. It is convenient to further classify the redundancies of type (2) and (3).

2. Redundancies of type 2 can be classified into
   (a) $F \in A$ produces a $int^F \neq int$ (a faulty output not equal to the true output) that is a specified output for all states that B can be in. int^F elicits the same response as int from B for all of these states and therefore F is redundant.
   (b) $F \in A$ produces an unspecified or *invalid* output int^F for the states B can be in and elicits the same response from B. In this case, one may elicit the same response from B or B might be moved to an invalid state that is equivalent to the true state resulting in redundancy.
   (c) A more complicated case of 2(a)/(b), where A produces a sequence of faulty outputs $int1^F$, .. $intN^F$ instead of $int1$, .. $intN$, such that the first output moves B into an invalid state that is *not* equivalent to the true state, but this state effectively becomes equivalent to the true state due to $int2^F$, ..$intN^F$.
3. Redundancies of type 3 can be classified into

(a) $F \in B$ requires a transition edge in B that cannot be justified for excitation/propagation to the primary output or next state lines.

(b) A transition edge that propagates $F \in B$ to the next state lines exists and the faulty state produced is a valid state. The faulty fault-free state pair in B possess a distinguishing sequence (which constitutes part of a test sequence), but this sequence cannot be produced at the outputs of A.

(c) Same as above, except that the faulty state that is produced is an invalid state.

Redundancies 2(a) and 2(b) are associated with single-vector don't care outputs of A. Of course, one may have multiple occurrences of faulty output vectors producing the same responses for a fault F of type 2(a) or 2(b). Redundancy 2(c) is associated with don't care output *sequences* (multiple vectors) of A. Redundancy 3(a) is associated the simple form of input don't care described in Section 3.1, where transition edges in B need not be specified. Redundancies 3(b) and 3(c) are associated with don't care input sequences to B.

5.2 A Synthesis Procedure for Irredundant Cascaded Machines

The procedure presented below represents a one-pass optimization for a cascade and eliminates a large number of redundancies in a cascade.

```
optimize-cascade( A, B ):
{
    output-expansion-1 ( A, B ) ;
    irredundant-1( A ) ;
    exploit-input-dc ( B, DC^A ) ;
    irredundant-1( B ) ;
}
```

Don't care outputs of A and don't care inputs to B are exploited. The procedure **irredundant-1()** uses the techniques described in the previous section to make a single machine irredundant in isolation.

Theorem 5.1 *The procedure* **optimize-cascade()** *produces a cascade $A \longrightarrow B$ that is irredundant for all type 1, type 2(a), type 3(a), type 3(b) and type 4 faults.*

Proof: Type 1 and type 4 faults cannot exist, since A and B are irredundant in isolation.

After the procedure **exploit-input-dc()** has been used, each remaining (specified) edge in the machine B, can be justified, by some input sequence to A. After B has been made prime and irredundant, we are guaranteed that at least one of the originally specified edges is a test vector in the combinational sense for any fault $F \in B$. That is, we have a vector that excites and propagates F to the primary outputs of B or the next state lines. This vector can be reached controlling A alone. Therefore, F cannot be a redundancy of type 3(a).

Next, consider redundancies of type 3(b). After the procedure **exploit-input-dc()** has been used on B with a complete don't care input sequence set, each pair of valid states remaining in B possess a distinguishing sequence that is *not* in the don't care input sequence set. This means that each pair of valid states can be distinguished via an input sequence to A. Therefore, if $F \in B$ produces a faulty fault-free state pair such that the faulty state is a valid state, then we have a distinguishing (test) sequence for F and F cannot be a redundancy of type 3(b). However, the same cannot be said of faulty states that are invalid and F may be a redundancy of type 3(c).

Redundancies of type 2(a) cannot exist because output expansion has been performed on A, using **output-expansion-1()**. A fault $F \in A$ can be initially propagated to the outputs of A or the next state lines. If all test vectors for F propagate F to the output lines alone and produce valid/specified faulty outputs (if even one vector produces an invalid output, F cannot be redundancy of type 2(a)), then because we have exploited the output don't cares for each transition edge in A, we are guaranteed that at least one of the vectors (edges) corrupted by F will elicit a different response for some state that B can be in. (By different response we mean that B goes to a different state or produces a different output). On the other hand, if F is propagated to the next state lines alone, then a corrupted vector will exist such that it produces a faulty state that can be distinguished from the true state *under the output don't care set*. This means we have a distinguishing input vector sequence (to A) such that the final faulty output necessarily elicits a different response from B or is an invalid/unspecified output. If F is propagated to both the outputs and the next state lines then for some test vector either the faulty output will directly elicit a different response from B or the faulty fault-free state pair will possess a distinguishing sequence that eventually elicits a different response from B. Thus, F is testable or is not a redundancy of type 2(a). **Q.E.D.**

Eliminating *type 1 SRFs* in a single machine required iterative optimization due to the existence of invalid states. In a cascade, we have a similar situation where eliminating type 2(b) and 3(c) redundancies (which might result in B moving to an invalid state) requires a two-pass optimization. This is because expanding the outputs of A to include invalid/unspecified outputs may introduce additional don't care input sequences to B.

```
irredundant-cascade( A, B ):
{
    for( iter = 1; iter ≤ 2; iter = iter + 1 ) {
        if ( iter = 1 ) output-expansion-1 ( A, B ) ;
        else output-expansion-2 ( A, B ) ;
        irredundant-1( A ) ;
        exploit-input-dc ( B, DC^A ) ;
        irredundant-2( B, DC^A ) ;
    }
}
```

The procedure **output-expansion-2()** is an enhanced version of **output-expansion-1()**. There are two enhancements corresponding to the don't cares for type 2(b) and type 2(c) redundant faults.

1. Given an optimized B, for each valid state, all the invalid states that are equivalent to this state are found. We might have a situation where for a particular transition edge in A, an output different from the edge's output places B in an invalid state that is equivalent to the true valid state. This output represents a don't care for the transition edge and is detected in **output-expansion-2()** (but not in **output-expansion-1()**). We also have the simpler situation of A producing a faulty output that was originally unspecified for the state(s) B is in, eliciting the same response from B. The output of the transition edge can be expanded to this unspecified combination.

2. Don't care output sequences are detected for A. The detection of these sequences is performed by checking if invalid states in B, that are not equivalent to valid states and reached by unspecified outputs from A, produce the same response in B due to the corruption of other transition edges in A. The corrupted outputs represent a don't care output sequence for edges in A. A 2-vector don't care sequence is shown below.

$$(o(e_1), o(e_2)) = (e_1 -> op, e_2 -> op) \bigvee$$
$$(e_1 -> op^F, e_2 -> op^F) \quad (1)$$

Current logic minimizers are restricted in their capability to exploit don't cares. Don't care output sequences of the form of Eq. 1 cannot be optimally exploited, other than by exhaustive search. Fortunately, these don't cares are not required in practice – we have not encountered a single occurrence of a type 2(c) redundancy in a cascade, even if only single-vector don't care outputs have been used.

The procedure **irredundant-2()** is also an enhancement on procedure **irredundant-1()**.
Irredundant-2() uses **eliminate-type1/2-SRFs()** with an additional don't care set at Step **A**. At Step **A**, we have

$$\mathbf{A}: FA^{DC} \mid fanin(q) = (q, v_1, .. v_m, iv_1, .. iv_n, ni_1, .. ni_l)$$

where $v_1, ..v_m$ and $iv_1, .. , iv_n$ are valid and invalid states respectively, that are equivalent to q when B is viewed in isolation, i.e. deemed completely controllable. $ni_1, ..ni_l$ are states *not* equivalent to q when B is viewed in isolation, but equivalent to q under the don't care set DC^A.

Theorem 5.2 *The procedure* **irredundant-cascade()** *results in an irredundant cascade.*

Proof: The procedure **irredundant-cascade()** is an enhanced version of the procedure **optimize-cascade()** and the arguments that type 1, type 2(a), type 3(a), type 3(b) and type 4 faults are testable hold here as well. We focus on possible redundancies of types 2(b), 2(c) and 3(c).

The procedure **output-expansion-2()** uses the additional don't care outputs for A corresponding to the invalid states in B that are equivalent to valid states and which are reached by outputs other than the transition edge outputs of A. Using these don't cares ensures that type 2(b) redundancies don't exist. The argument is similar to the argument of Theorem 5.1 for the type 2(a) redundancy. A fault $F \in A$ will immediately or eventually produce an invalid/unspecified output such that the invalid output elicits a different response from B. If B is moved to a faulty invalid state we are guaranteed that the invalid state is not equivalent to the true state. Thus, F is testable or is not a redundancy of type 2(b).

Redundancies of type 2(c) are associated with don't care output sequences for A. That is, it does not matter if A asserts one particular sequence or another due to its constrained observability. If the don't care sequences corresponding to Eq. 1 are exploited in the output expansion procedure, we are guaranteed that the corrupted sequence does not elicit the same response as the true one from B.

Finally, we consider redundancies of type 3(c). The additional don't care set at Step **A** in **eliminate-type1/2-SRFs()** will guarantee, after B has been made prime and irredundant, that any faulty faulty-free state pair that is produced due to a fault F, regardless of whether the faulty state is valid or invalid, will possess a distinguishing sequence not in DC^A. This means that the pair can be distinguished from the inputs of A and F cannot be a redundancy of type 3(c). **Q.E.D.**

6 Fully Testable Interacting Finite State Machines

Interacting finite state machines are common in industrial chip designs. In Figure 6, an example sequential circuit composed of three interacting finite state machines was shown. In this section, we describe *iterative* optimization strategies for the synthesis of irredundant interacting finite state machines.

The don't care sets associated with a set of interacting FSMs are essentially the same as those in a cascade. At any given set of intermediate lines or latches that are not observable/controllable we have don't care input and output sequences. We can view an arbitrary set of interacting machines as several occurrences of individual cascades and use the don't care sets required for synthesizing irredundant cascades iteratively to eliminate all redundancies in the circuit.

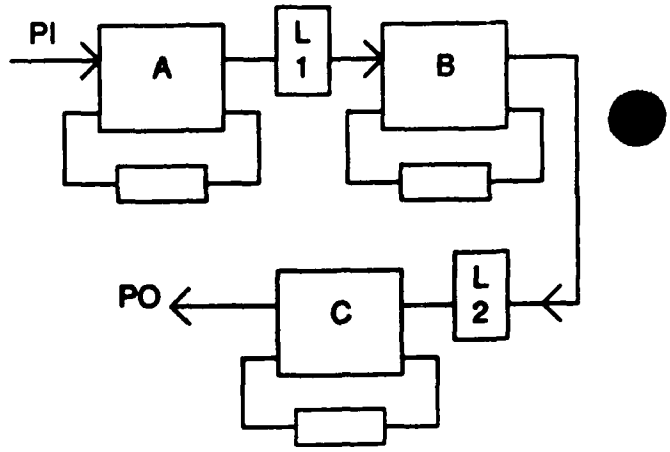


Figure 11: A Cascade Chain

We first focus on the cascade chain of Figure 11. There are two individual cascades corresponding to $A \longrightarrow B$ and $B \longrightarrow C$. It is easy to see that if one optimizes $A \longrightarrow B$ first and then $B \longrightarrow C$, then we obtain the entire set of don't care input sequences to C, but not vice versa. This is because optimizing B with its don't care input sequences may produce additional don't care input sequences for C. Thus, an appropriate order of optimization of individual cascades is required. However, optimizing $A \longrightarrow B$ before $B \longrightarrow C$ may result in missing some don't care outputs for output expansion, as illustrated in Figure 12. We have fragments of the State Transition Graphs corresponding to A, B and C in Figure 12. We would raise the outputs of the edge in A only if we optimized $B \longrightarrow C$ before $A \longrightarrow B$.

We have thus a conflict between the order of optimization of the individual cascades, if we wish to make use of all the don't care sets. This conflict is resolved in the cascade chain case quite simply, by an optimization $A \longrightarrow B$, $B \longrightarrow C$ and $A \longrightarrow B$. The case of Figure 6, a more general case where global feedback exists, is more complicated and requires iteration to convergence of the three individual cascades, $A \longrightarrow B$, $B \longrightarrow C$, $C \longrightarrow A$. Iteration to convergence is required because the global feedback may result in, say additional don't care inputs to B after the don't care inputs to A have been exploited, which in turn are dependent on the output don't cares of C and so on.

Given an arbitrary interconnection of FSMs, the elimination of all redundancies entails the optimization of every path from the primary inputs to the primary outputs using the input don't care sets. Similarly, every reverse path from the primary outputs to the primary inputs has to be optimized for output don't cares. Iterative optimization to convergence is required in the case of feedback paths. If this is done and the machines are all irredundant in isolation, the interconnection will be irredundant. Any fault, F, in any machine M, will possess a test sequence at M's inputs whose effects can be observed at M's outputs. Exploiting don't care input sequences from the primary inputs outward to the primary outputs ensures that this test sequence can be produced at the inputs of M. Exploiting the don't care outputs from the primary outputs inward to the primary inputs ensures that the effect of the test sequence will be propagated to the primary outputs.

7 Results

In this section, we present some preliminary results obtained using the synthesis procedures described in Section 5 and 6.

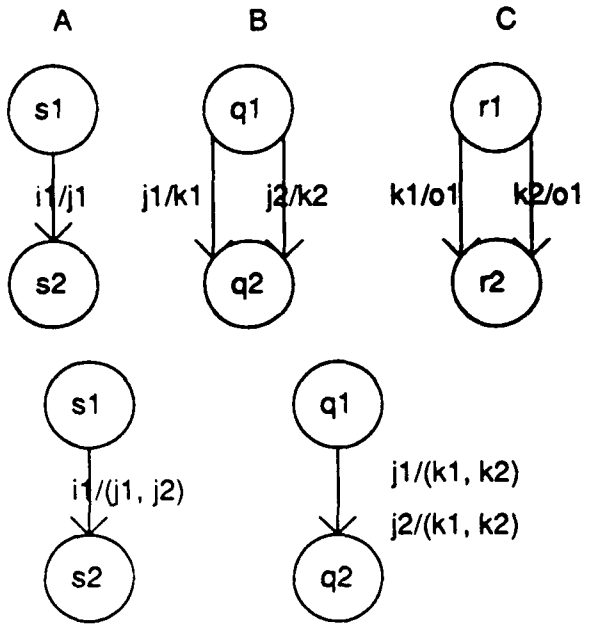


Figure 12: Order of Optimization for Don't Care Outputs

Ex	pi	po	ma	il	la	#states			#lit
						M1	M2	M3	
ex1	[illegible]	19	2	14	25	20	48	-	704
ex2	11	9	2	8	16	10	16	-	338
ex3	[illegible]	2	2	4	12	16	16	-	186
ex4	8	11	3	13	28	20	32	19	926
ex5	8	21	3	16	32	20	16	48	772

Table 1: Statistics of Examples

Intensive optimization is necessary to obtain fully testable designs. If this optimization can be carried out, then the synthesized machine(s) will occupy minimal area. There is no area/performance overhead associated with this procedure. However, the CPU time requirements have to be evaluated.

We chose some examples in the MCNC 1987 Logic Synthesis Workshop as test cases, whose statistics are given in Table 1. These machines were interconnected in various ways. In Table 1, the number of primary inputs (pi) and primary outputs (po), the number of separate machines (ma) and the number of states in each machine in the circuit (#states) are indicated for each example. The number of intermediate, non-observable/non-controllable lines (il) and the total number of literals (#lit) after state assignment using MUSTANG [6] and multi-level combinational optimization using MIS [3] are also given. The total number of latches (la) corresponds to a minimum bit encoding for each machine and the pipeline latches in the intermediate lines.

The program STALLION [9] was used initially to generate tests for the original circuits. The results are given under the column ORIGINAL. The time in CPU minutes required for test generation (TPG time), the fault coverage obtained by STALLION (fcov), and the original literal count for each circuit (lit) are indicated. The circuits were optimized using the various don't cares described in the previous sections. The CPU time required for logic optimization (logic time), test generation (TPG time), fault coverage obtained by STALLION (fcov) and the final literal counts (lit) are indicated under the column OPTIMIZE for each circuit.

All the circuits have been reduced in complexity and made irredundant. STALLION either aborted or identified the undetected faults in the original circuits to be redundant. While the CPU times required for logic optimization are typically larger than the test generation times, the alternative of explicitly identifying redundancies in the original circuits via test generation would expend considerably more CPU time. Redundant lines corresponding to redundant stuck-at faults can only be removed one at a time. Futhermore, removing a redundant line may introduce new redundancies and so *all* faults have to be checked for redundancy on *each* removal. Thus, the implicit elimination of redundancies via the use of don't care sets represents a much more efficient approach to the synthesis of irredundant interacting sequential machines.

Ex	ORIGINAL			OPTIMIZE			
	TPG time	fcov	lit	logic time	TPG time	fcov	lit
ex1	18.1m	93.1	704	42m	17.4m	100	641
ex2	4.5m	91.6	338	14m	4.6m	100	241
ex3	4.8m	95.7	186	11m	4.7m	100	151
ex4	21.2m	91.4	926	45m	18.3m	100	781
ex5	25.1m	92.5	772	38m	22.4m	100	642

Table 2: Results using Don't Care Exploitation Algorithms

8 Conclusions

In this paper, we explored the relationships between redundant logic and don't care conditions in sequential circuits. Redundancies in non-scan sequential circuits may be testable from a combinational viewpoint, but may produce a faulty State Transition Graph (STG) that is equivalent to the STG of the true machine.

We presented a classification of redundant faults in sequential circuits composed of single or interacting finite state machines. Don't care sets can be defined for each class of redundancy and optimally exploiting these don't care conditions results in the implicit elimination of any such redundancies in a given circuit. In cascaded and interconnected sequential circuits, sequential don't cares are required to eliminate redundancies.

We presented preliminary experimental results which indicate that medium-sized irredundant sequential circuits can be synthesized with no area overhead and within reasonable CPU times by exploiting these don't cares.

9 Acknowledgements

The interesting discussions with Pranav Ashar, Robert Brayton, Kurt Keutzer and Alberto Sangiovanni-Vincentelli on sequential machine testability are acknowledged. This work was supported in part by the the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

References

[1] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level logic minimization using implicit don't cares. In *IEEE Transactions on CAD*, pages 723–740, June 1988.

[2] D. Brand. Redundancy and don't cares in logic synthesis. In *IEEE Transactions on Computers*, pages 947–952, October 1983.

[3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: a multiple-level logic optimization system. In *IEEE Transactions on CAD*, pages 1062–1081, November 1987.

[4] S. Devadas, H-K. T. Ma, and A. R. Newton. Easily Testable PLA-based Finite State Machines. In *Proc. of 19th Fault Tolerant Computing Symposium*, June 1989.

[5] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant sequential machines via optimal logic synthesis. In *Electronics Research Laboratory Memorandum M88/52*, University of California, Berkeley, August 1988.

[6] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multi-level logic implementations. In *IEEE Transactions on CAD*, pages 1290–1300, December 1988.

[7] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Synthesis and optimization procedures for fully and easily testable sequential machines. In *Proc. of International Test Conference*, pages 621–630, September 1988.

[8] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. In *IEEE Transactions on Computers*, pages 215–222, March 1981.

[9] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test generation for sequential circuits. In *IEEE Transactions on CAD*, pages 1081–1093, October 1988.

[10] M. C. Paul and S. H. Unger. Minimizing the number of states in incompletely specified sequential circuits. In *IRE Transactions on Electronic Computers*, pages 356–357, September 1959.

[11] J. P. Roth. Diagnosis of automata failures: a calculus and a method. In *IBM journal of Research and Development*, pages 278–291, July 1966.